Algorithmes avancés : Problème du voyageur de commerce

Kevin BASCOL, Kevin LAOUSSING, Nicolas REYNAUD

Sommaire

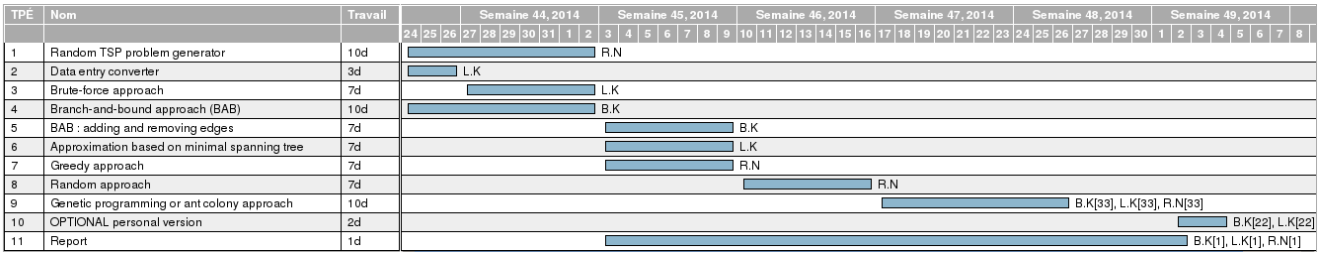
1 Introduction

Le problème du voyageur de commerce, consiste en la recherche d'un trajet minimal permettant à un voyageur de visiter n villes. En règle générale on cherche à minimiser le temps de parcours total ou la distance totale parcourue.
Il existe plusieurs approche pour résoudre ce problème, parmi lesquelles on peut distinguer les approches donnant des résultats exactes et les approches donnant des résultats approximatives. L'objectif de notre projet est d'implémenter ces différentes approches, de pouvoir les tester, de pouvoir analyser les résultats et de déterminer quel est le meilleur compromis entre temps de résolutions et "exactitude" de la solution.

2 Organisation du projet

2.1 Planning

2.1.1 Diagramme de gantt



2.2 Langage utilisé et outil de développement

Nous avons choisi d'implémenter tous nos algorithmes en langage C pour sa rapidité. En effet les graphes étant chargés en mémoire et le langage C permettant un accès rapide à celle-ci, ce qui rendrait le programme plus performant en terme de temps d'exécution. Nous avons utilisé le logiciel git/github pour gérer le développement de nos codes du projet.

3 Algorithmes de résolution de problème du voyageur de commerce

3.1 Structure de donnée

La structure de donnée du graphe est très simple. Elle consiste en un accès direct à chaque nœud. Le nombre de nœud est également présent en accès direct. Chaque nœud est ensuite représenté par :

- Un nom N
- Une coloration (Indique si le nœud à été visité ou non)
- Le nombre de sous-nœud
- Un tableau contenant tous les successeurs du nœud avec le nœud d'indice i égale au i-ème nœud.
- Un tableau contenant le coût du chemin entre le nœud N courant et le nœud présent en case i.

La structure de donnée de la solution a, quant à elle, évolué au fil du temps. Passant à la base d'une liste de nom de nœud à une liste de nœud. Elle contient également le nombre de nœuds présents dans la solution, et le coût du chemin présent dans la solution.

3.2 Algorithme donnant une solution optimale

3.2.1 Force brute

3.2.1.1 Résumé de l'algorithme

L'algorithme de brute force consiste à explorer toutes les solutions du graphe : c'est-à-dire d'évaluer tous les cycles possibles d'un graphe et de retenir la solution optimale. L'avantage est qu'il est simple à implémenter, et qu'il donne la solution optimale. Mais son plus grand inconvénient est sa complexité en temps :
$O(n!) \approx O(\exp n)$.

3.2.1.2 Commentaires sur l'implémentation

L'algorithme a été implémenter en une fonction récursif.

3.2.2 Séparation et évaluation

3.2.2.1 Résumé de l'algorithme

L'algorithme de séparation et évaluation consiste en un parcours de l'arbre de tous les chemins possible dans le graphe des villes. Mais si le chemin menant à un nœud possède un poids total supérieur à la meilleurs solution trouvée jusque là, alors il ne parcourra pas les sous-arbres de ce nœud.
La complexité de cet algorithme est donc, au pire, la même que celle de l'algorithme par force brute (O(n!)) mais est grandement améliorée grâce à l'élagage des branches trop lourdes.

3.2.2.2 Commentaires sur l'implémentation

Cet algorithme a été implémenté récursivement, méthode plus lourde mais aussi plus intuitive. Lors de la programmation de cet algorithme j'ai malheureusement oublié de faire le retour permettant l'élagage des branches trop lourdes. Cette erreur m'a fait perdre beaucoup de temps à chercher des méthodes pour accélérer le programme alors qu'il ne manquait que 2 lignes de codes pour obtenir un programme extrêmement plus rapide.

3.2.3 Séparation et évaluation avec retrait d'arêtes

3.2.3.1 Résumé de l'algorithme

L'algorithme de séparation et évaluation avec retrait d'arêtes consiste en un parcours d'arbre. Celui-ci est un arbre binaire, chaque nœud est la matrice des arêtes du graphe, le fils gauche correspond à la matrice résultante du passage par une arête choisie, le fils droit représente la matrice résultante du retrait de cette arête. Le parcours s'arrête au moment où la fonction d'évaluation retourne une valeur supérieur à celle de la meilleure solution trouvée.

3.2.3.2 Commentaires sur l'implémentation

Je n'ai malheureusement pas réussi à implémenter un programme fonctionnel pour cet algorithme avant la fin du temps imparti. Ceci est en partie dû à la compréhension difficile de l'algorithme, mais aussi à la réalisation chronophage des autres projets universitaires. J'ai tenté une version récursive mais me suis embourbé au milieu d'une double récursion.

3.3 Algorithme donnant une solution approximative

3.3.1 Approche cupide

3.3.1.1 Résumé de l'algorithme

L'algorithme de l'approche cupide commence par se positionner sur un nœud du graphe. A partir de là, l'algorithme passe au nœud suivant non visité en regardant l'arête avec le poids minimum. Et ainsi de suite jusqu'à avoir parcouru tout les nœuds du graphe.

3.3.1.2 Commentaires sur l'implémentation

Le code de ce programme étant assez simple, aucun soucis particulier n'est à notifier dans ce code.

3.3.2 Approche aléatoire

3.3.2.1 Résumé de l'algorithme

L'approche aléatoire est très simple. L'algorithme se place sur un nœud aléatoire du graphe. A partir de là, l'algorithme passe au nœud suivant non visité en choisissant une arête de manière aléatoire. Jusqu'à revenir à sont point de départ. L'algorithme est lancé une première fois. Puis est lancé 100 fois en comparant les résultats entre eux. A chaque fois le chemin avec le coup le moins élevé est choisit.

3.3.2.2 Commentaires sur l'implémentation

Le code de ce programme étant assez simple, aucun soucis particulier n'est à notifier dans ce code. Cependant très rapidement j'ai fais le choix de séparer le code en deux méthodes, dont une avec un paramètre permettant de donner la coloration a utiliser pendant le parcours du graphe. Tout ceci dans le but de me resservir de la l'approche aléatoire dans la génération de la population dans l'approche génétique.

3.3.3 Approche avec l'arbre couvrant de poids minimal

3.3.3.1 Résumé de l'algorithme

L'algorithme utilise l'algorithme de Prim ou de Kruskal (dans notre projet, nous avons utilisé l'algorithme de Prim) pour construire un arbre couvrant de poids minimal reliant tout les noeuds du graphe. Une fois que l'arbre couvrant est construite, la solution du problème est donnée par un parcours préfixe de l'arbre.
L'algorithme est efficace si le distance entre les noeuds respectent la règle de Pythagore. Nous verrons dans le section 4 que cette algorithme donne une solution approximative plus proche de la solution optimale lorsque nous utilisons des graphes du site donnée dans le sujet.

3.3.3.2 Commentaires sur l'implémentation

Nous avons utilisé l'algorithme de Prim parce qu'il était plus simple à implémenter. En effet l'algorithme de Kruskal nécessitait d'énumérer toute les arêtes du graphe. Cette opération aurait été trop coûteuse avec la structure de donnée que nous nous sommes donnés.

3.3.4 Approche génétique

3.3.4.1 Résumé de l'algorithme

L'algorithme de l'approche génétique passe par plusieurs sous méthode.
Pour commencer une population N de solution est générée.
Cette population est alors modifiée, elle évolue.
l'évolution consiste à trier la population par ordre croissant, à partir de là, un pourcentage d'élite est gardé.
Les élites peuvent, avec un faible pourcentage de chance muter. La mutation se traduit par l'inversion de 2 nœuds dans la solution élite.
Le reste de la population subit alors des croisements et des mutations. Les croisement se font sur deux parents. Les parents sont choisit, soit parmi les élites avec un faible pourcentage de chance. Soit parmi une sélection aléatoire de parent parmi les éléments triées.
Le tournois choisit simplement alors la meilleurs solution (Celle avec le cout le plus faible) parmi la sélection aléatoire.
Le croisement peut alors commencer. Le croisement comporte alors 2 étapes :

- On commencer par copier une partie de la solution du parent numéro 1. Cette partie est choisit aléatoirement.
- On complète ensuite la solution fils obtenu en complétant celle ci par les nœuds présent dans le second parent. En conservant l'ordre présent dans le parent 2.

Le fils ainsi obtenu à alors un faible chance de muter.
L'évolution est ensuite répétée X fois. En prenant comme nouvelle entré la sortie de l'évolution précédente.
Pour finir l'algorithme choisit la meilleurs solution (Celle avec le plus faible coût) après évolution .

3.3.4.2 Commentaires sur l'implémentation

Ce code étant assez compliqué, il m'a fallu pas mal de temps pour bien comprendre son fonctionnement. Qui plus est ayant directement cherché une méthode peu couteuse en mémoire (notamment en réutilisant les tableaux au maximum) de nombreuses erreurs ont été commises au début du code.

Par la suite, le seule autre problème que j'ai pu avoir fut sur le paramétrage. Étant donné le "grand" nombre de paramètre il à été dur de les déterminer.

A noter : Les paramètrages actuels sont idéaux pour avoir un temps de réponse rapide et juste pour des graphes de 2 à 10 nœuds. Pour avoir de meilleurs résultat sur des graphes possédant un plus grand nombre de noeud, il faudrait augmenter le nombre de solution générées au départ et également le nombre d'évolution. Par exemple, pour des graphes de 15 à 20 nœuds il faudrait environs triplé le nomre d'évolution.

4 Commentaire général sur le code

Les conventions on été respectée.
Qui plus est, nous avons mis un point d'honneur à avoir un code des plus propre possible sur les allocations et libérations de mémoire.
Voici le retour de valgrind lancé avec les options : "-v –leak-check=full" sur un graphe à 5 nœuds.

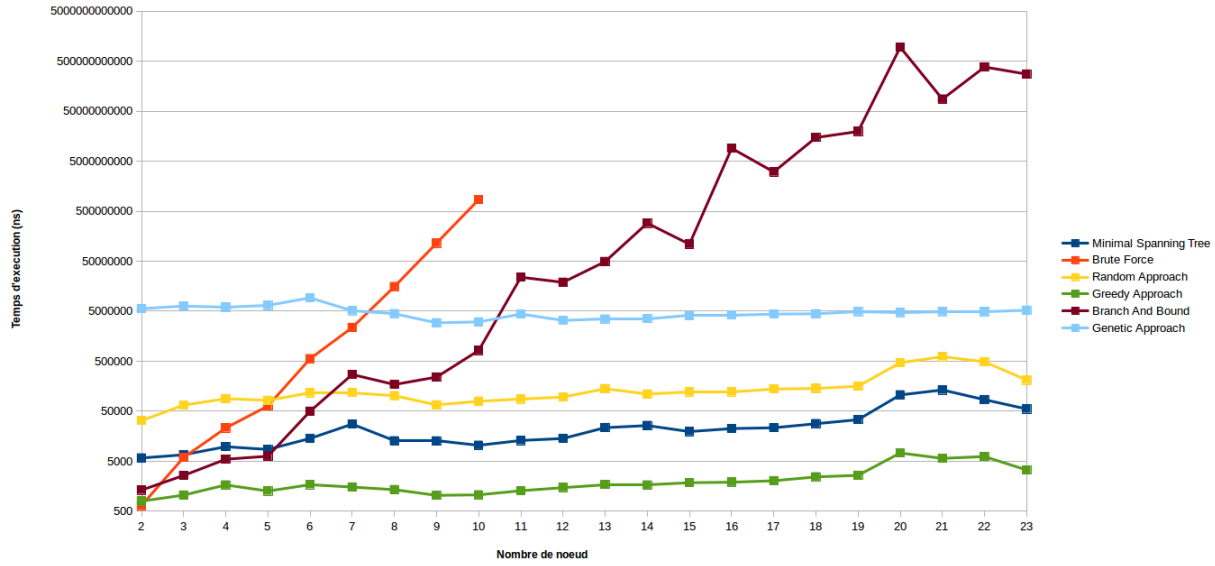
```
==6610== HEAP SUMMARY:
==6610==     in use at exit: 0 bytes in 0 blocks
==6610==   total heap usage: 29,231 allocs, 29,231 frees, 1,168,096 bytes allocated
==6610==
==6610== All heap blocks were freed -- no leaks are possible
==6610==
==6610== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==6610== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5 Etudes d'efficacités des algorithmes

Cette section présente les résultats de nos tests des algorithmes que nous avons implémenté. Elle se présente en deux parties, car nous avons fait des tests sur des graphes générés aléatoirement par notre programme TSPGenerateur et des tests issues de la base de donnée du site mentionné dans le sujet. En effet les performances de nos algorithmes varient en fonction des graphes générés aléatoirement ou non.

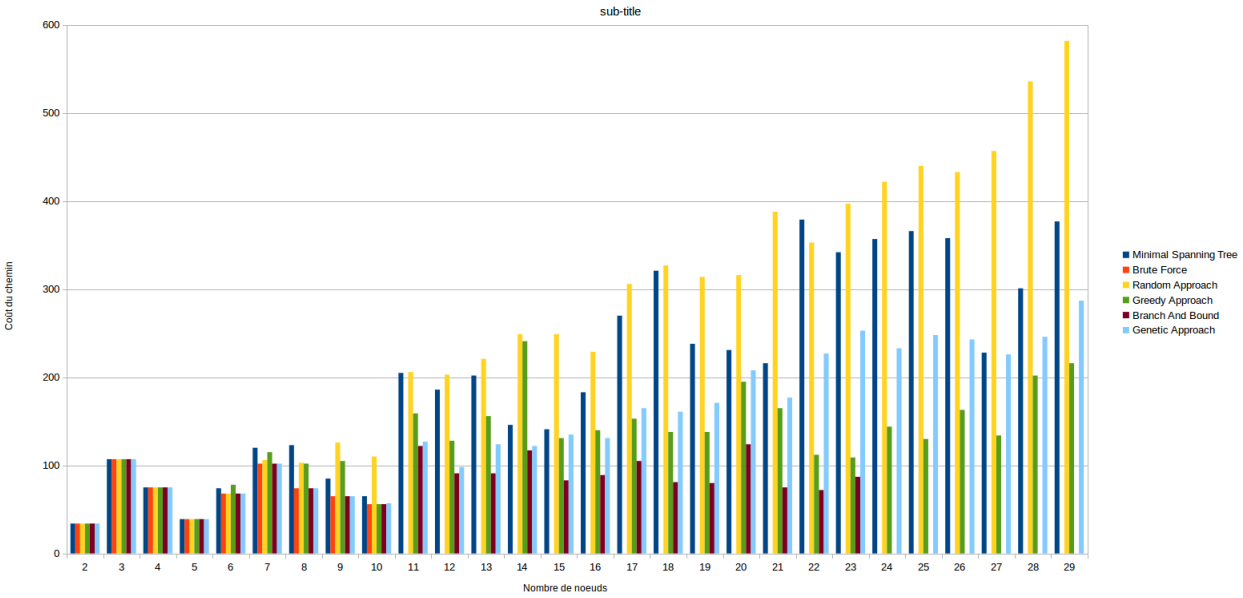
5.1 Graphes générés aléatoirement

5.1.1 Tests temps d'exécution



On peut observer qu'à partir de graphes de 8 nœuds, l'algorithme par force brute se termine bien après les autres algorithmes, ce qui est un résultat attendu puisqu'il a la complexité la plus grande.
L'algorithme de séparation et évaluation, quant à lui, devient le plus lent à partir de 11 nœuds. Sa courbe irrégulière est causé par l'élagage qui retire plus ou moins de possibilité selon le graphe.
Les autres algorithmes présentent, à partir de 11 nœuds, une durée plus modérée que les deux précédents et une augmentation de la durée d'exécution plus régulière.

5.1.2 Tests coût



Ici les algorithmes par force brute et de sélection et évaluation donnent toujours une solution avec un poids optimal, ils serviront de repère.

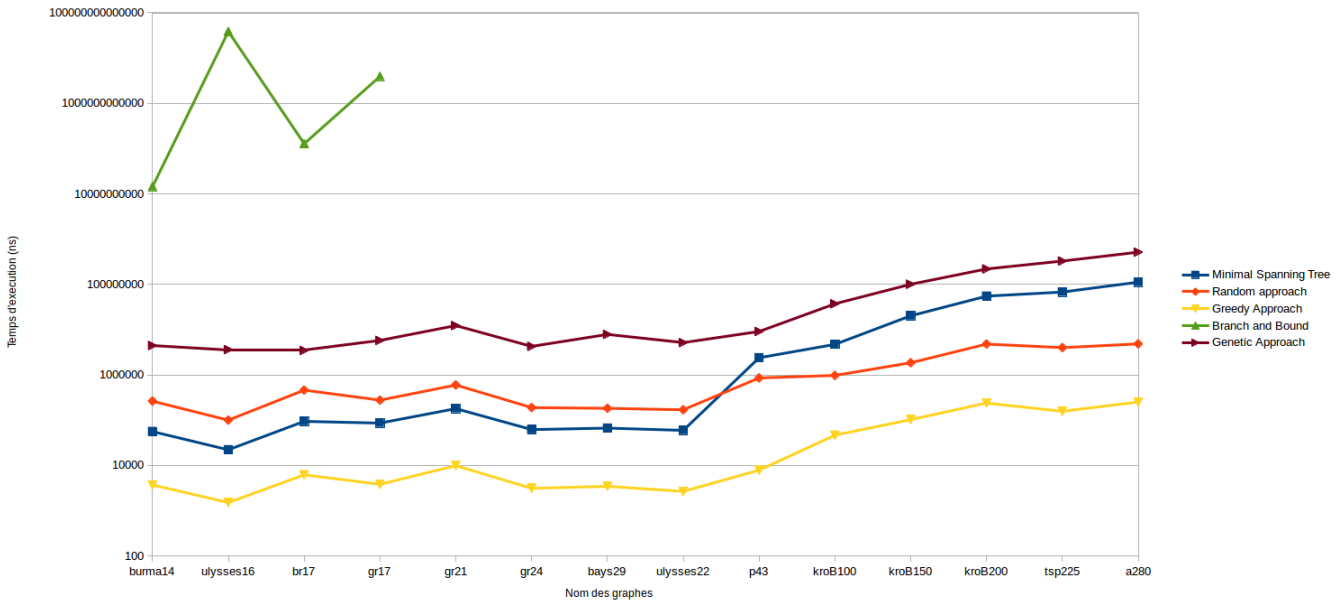
Cet histogramme nous apprend que jusqu'à 7 nœuds tous les algorithmes fournissent une solution quasi-optimale.
A partir de 8 nœuds l'algorithme aléatoire donnera le plus souvent la pire réponse des tests. L'algorithme de l'arbre couvrant donnera la plupart du temps une solution comparable à celle de l'approche aléatoire.
Jusqu'à 21 nœuds, les algorithmes cupide et génétique présentent des résultats comparable mais de plus en plus éloigné de la solution optimale. à partir de 22 nœuds, l'approche cupide est la plus proche de la solution optimale, alors que l'approche génétique s'en éloigne.

5.1.3 Conclusion

Les algorithmes par force brute et de sélection et évaluation sont les seuls donnant toujours une solution optimale. Mais ils présentent aussi le coût en temps extrêmement plus important que les autres, en effet force brute devient inutilisable à environ 10 nœuds et sélection à environ 23 nœuds. Parmi les autres algorithmes, l'approche aléatoire est clairement à éviter n'étant pas la plus rapide et étant la moins précise.
L'approche par arbre couvrant est plus rapide que l'approche aléatoire mais est toute aussi imprécise et serait donc à éviter aussi.
L'approche génétique est relativement précise jusqu'à 17 nœuds mais aussi l'une des plus lente et n'est donc pas la meilleure option.
L'approche cupide, quant à elle, est la plus rapide de toutes, mais aussi, la plupart du temps, la plus précise.
Donc pour des graphes aléatoires de 2 à 19 nœuds on utiliserait l'algorithme de sélection et évaluation donnant une valeur optimale, et à partir de 20 nœuds on utiliserait l'algorithme cupide.

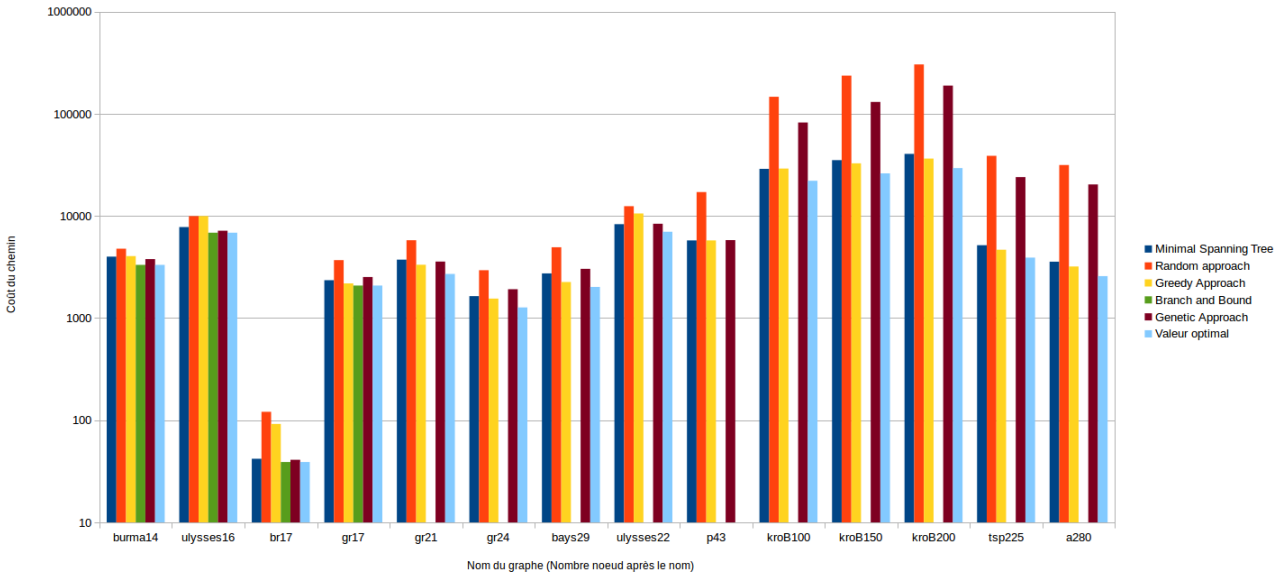
5.2 Graphes en ligne

5.2.1 Tests temps d'exécution



On observe que l'algorithme de sélection et évaluation présente une durée d'exécution extrêmement supérieur aux autres algorithmes. L'algorithme cupide est le plus rapide. de 14 à 29 nœuds l'approche par arbre couvrant est la 2ème plus rapide mais de 29 à 280 nœuds elle échange sa place avec la 3ème plus rapide qui est l'approche aléatoire. L'algorithme génétique est le plus lent des approximatifs et présente une augmentation de la durée d'exécution bien plus importante que les autres pour les graphes de plus de 100 nœuds.

5.2.2 Tests coût



Les tests de coût avec les graphes de la base de donnée du site étant relativement long pour l'algorithme de séparation et d'évaluation, nous avons décidé de faire les tests de coût de celui-ci sur des graphes dont le nombre de nœuds sont inférieur à 20. Pour les graphes dont le nombre de nœuds sont supérieur à 20 nous testons uniquement les algorithmes donnant des solutions approximatives que nous comparons avec les solutions (optimales) données par le site.
Sur les graphes de 14 à 17 nœuds, nous observons peu de variation entre les solutions des différents algorithmes. L'algorithme de séparation et évaluation donne des solutions optimales.
Sur les graphes de grandes tailles (nombre de nœuds supérieur à 100) les variations entre les solutions sont plus marquées : de manière générale, les algorithmes d'approche aléatoire et d'approche génétique donnent des solutions plus éloignées de la solution optimale. Cependant, nous savons que l'approche génétique est "calibré" pour les graphes de taille inférieur à 10 nœuds ; et pour obtenir des solutions plus proches de la solution optimale, il faudrait faire plus d'itération dans l'algorithme d'approche aléatoire. L'approche avec l'arbre couvrant de poids minimal et l'approche cupide sont des algorithmes approximations qui donnent des solutions relativement correctes et sont par conséquent plus intéressant pour résoudre des graphes de grandes tailles.

5.2.3 Conclusion

L'algorithme de sélection et évaluation donne une solution optimale mais est extrêmement lent sur ces graphes, il est donc peu intéressant.
L'approche aléatoire est une des plus rapides mais aussi la moins précise, elle serait donc aussi à éviter.
L'algorithme génétique étant aussi peu précis que l'aléatoire mais aussi beaucoup plus lent il serait à éviter également.
Pour ce qui est des algorithmes cupide et d'arbre couvrant, leur précision est comparable et bien meilleurs que les 2 autres. Mais l'approche par l'arbre couvrant étant plus lente (même si 0.1 seconde pour 280 nœuds reste tout à fait honorable) la meilleure approche est donné par l'algorithme cupide.

6 Conclusion

La construction des graphes peut avoir un important impacte sur la qualité d'un algorithme. En effet un graphe respectant la règle de Pythagore, contrairement à un graphe aléatoire, permet à l'approche de l'arbre couvrant d'être plus précise mais oblige l'approche par sélection et évaluation à parcourir plus de solution et donc la condamne à être beaucoup plus lente.

Donc pour des petits graphes aléatoire l'approche par sélection et évaluation permet d'obtenir rapidement une solution optimale. Pour des graphes respectant la règle de Pythagore l'approche par arbre couvrant est une solution rapide et précise. Cependant l'algorithme cupide fourni le meilleur ratio rapidité/fiabilité et est donc l'approche que nous avons codé à privilégier.

7 Annexes

7.1 DataConverter

Ce programme permet de générer de convertir les donnée XML du site mentionné dans le sujet dans un format que notre programme peut lire. Ce format est le même que celui générer par le TSPGenerateur.

7.2 TSPGenerateur

Permet de générer des graphes aléatoires, pour en connaitre le fonctionnement voir.

```
./TSPGenerator -h
```

Les fichiers de sortie sont dans le format défini par notre groupe.